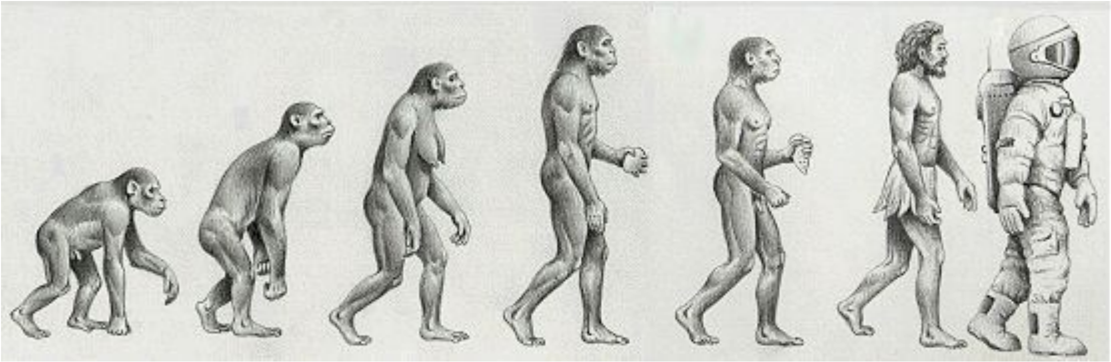
Pre-announcements

- Info session at Kresge Library at 6:30 PM tomorrow 1/31 if you want to help out with autonomous underwater vehicles.

- No experience needed all are welcome.



CS61B: 2019

Lecture 4: Node Based Lists

- From IntList to SLList
  - The private keyword
  - Nested classes
  - Recursive private helper methods
  - Caching
  - Sentinel nodes

From IntList to SLList

Last Time in 61B: Recursive Implementation of a List

```java
public class IntList {
   public int first;
   public IntList rest;

   public IntList(int f, IntList r) {
      first = f;
      rest = r;
   }
   ...
```

While functional, “naked” linked lists like the one above are hard to use.

- Users of this class are probably going to need to know references very well, and be able to think recursively. Let’s make our users’ lives easier.

Improvement #1: Rebranding and Culling

```
public class IntNode {
   public int item;
   public IntNode next;

   public IntNode(int i, IntNode n) {
      item = i;
      next = n;
   }
}
```

IntNode is now dumb, has no methods. We will reintroduce functionality in the coming slides.

Not much of an improvement obviously, but this next weird trick will be more impressive.

Improvement #2: Bureaucracy

```java
public class IntNode {
    public int item;
    public IntNode next;

    public IntNode(int i, IntNode n) {
        item = i;
        next = n;
    }
}
```

```java
IntList X = new IntList(10, null);
SLList Y = new SLList(10);
```

SLList is easier to instantiate (no need to specify `null`), but we will see more advantages to come.

```java
public class SLList {
    public IntNode first;

    public SLList(int x) {
        first = new IntNode(x, null);
    }
    ...
}
```

IntNode is now dumb, has no methods.

Next: Let's add addFirst and getFirst methods to SLList.

The Basic SLList and Helper IntNode Class

```java
public class SLList {
    public IntNode first;

    public SLList(int x) {
        first = new IntNode(x, null);
    }

    public void addFirst(int x) {
        first = new IntNode(x, first);
    }

    public int getFirst() {
        return first.item;
    }
}
```

```java
public class IntNode {
    public int item;
    public IntNode next;

    public IntNode(int i, IntNode n) {
        item = i;
        next = n;
    }
}
```

Example usage:

```java
SLList L = new SLList(15);
L.addFirst(10);
L.addFirst(5);
int x = L.getFirst();
```

SLLists vs. IntLists

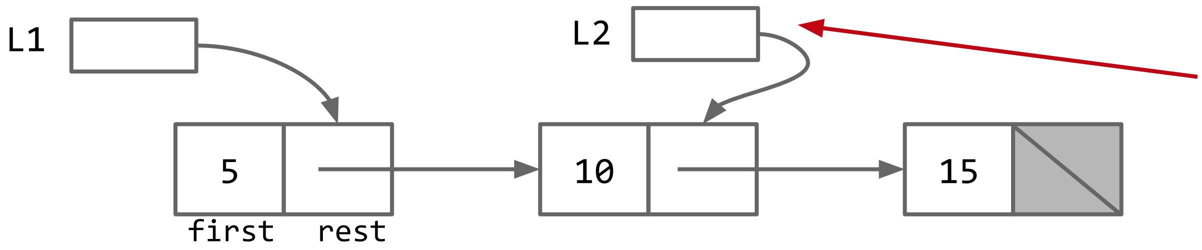
```
SLList L = new SLList(15);
L.addFirst(10);
L.addFirst(5);
int x = L.getFirst();
```

```
IntList L = new IntList(15, null);
L = new IntList(10, L);
L = new IntList(5, L);
int x = L.first;
```

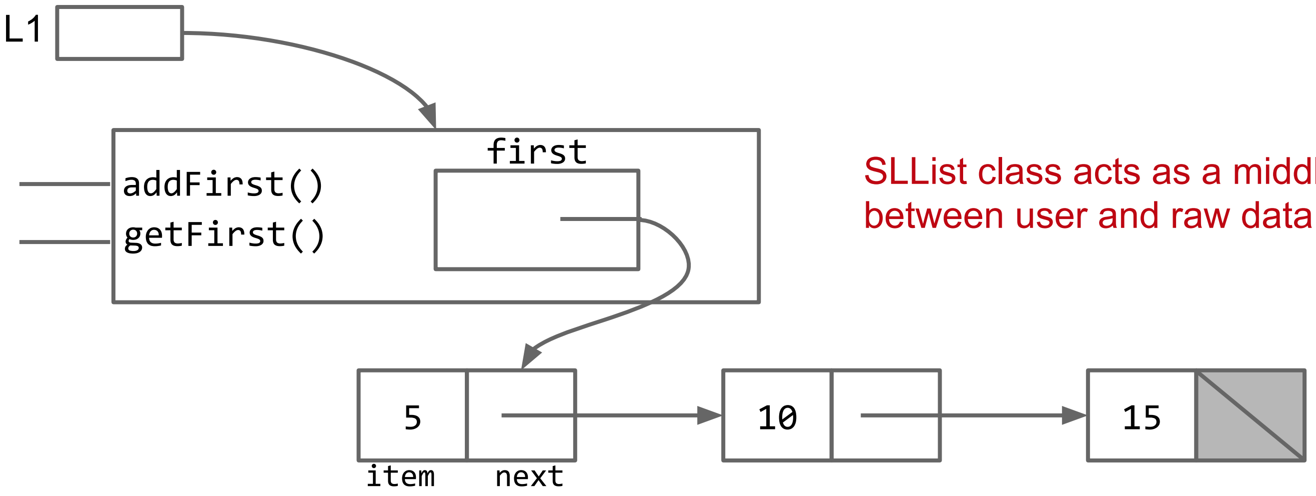
While functional, “naked” linked lists like the `IntList` class are hard to use.

- Users of `IntList` are need to know Java references well, and be able to think recursively.
- `SLList` is much simpler to use. Simply use the provided methods.
- Why not just add an `addFirst` method to the `IntList` class? Turns out there is no efficient way to do this. See exercises in lectureCode repository.

Naked Linked Lists (IntList) vs. SLLists



Naked recursion: Natural for IntList user to have variables that point to the middle of the IntList.

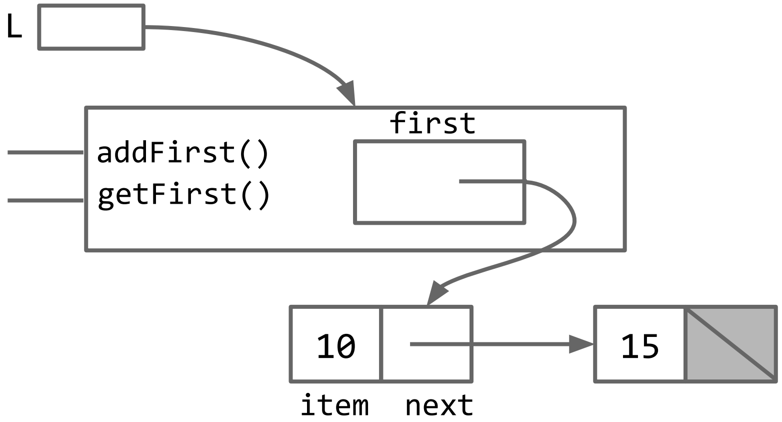


SLList class acts as a middle man between user and raw data structure.

Public vs. Private Nested Classes

The SLList So Far

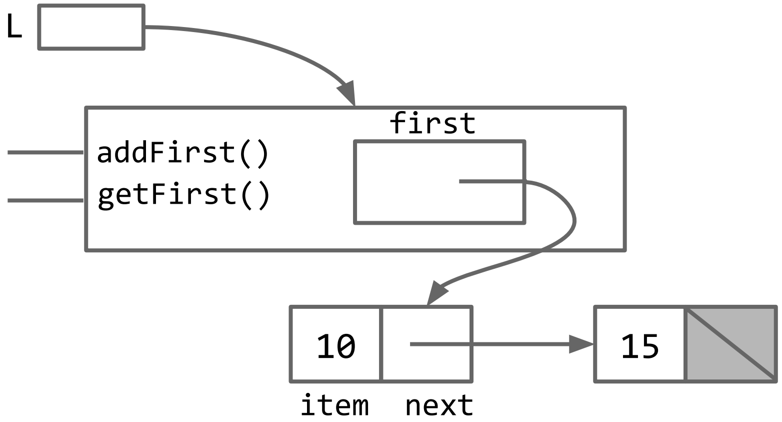
```java
public class SLList {
   public IntNode first;

   public SLList(int x) {
      first = new IntNode(x, null);
   }

   public void addFirst(int x) {
      first = new IntNode(x, first);
   }

   ...
}
```




```java
SLList L = new SLList(15);
L.addFirst(10);
```

A Potential SLList Danger

```java
public class SLList {
   public IntNode first;

   public SLList(int x) {
      first = new IntNode(x, null);
   }

   public void addFirst(int x) {
      first = new IntNode(x, first);
   }

   ...
}
```

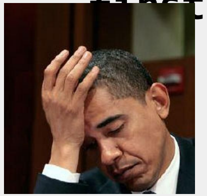


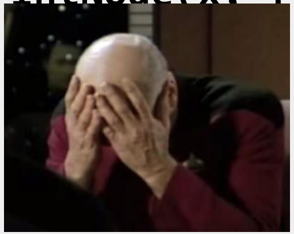
Users of our class might be tempted to try to manipulate our secret IntNode directly in uncouth ways!

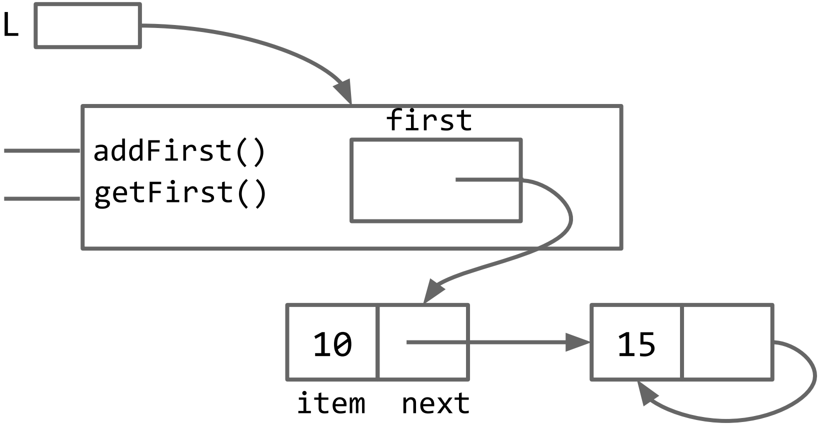
```java
SLList L = new SLList(15);
L.addFirst(10);
L.first.next.next = L.first.next;
```

A Potential SLList Danger

```
public class SLList {
    public IntNode first;

    publ                  x) {
        f                 tNode
    }

    public void addFirst(int x) {
        first = new IntNode(x, first);

}
```







Users of our class might be tempted to try to manipulate our secret IntNode directly in uncouth ways!

```
SLList L = new SLList(15);
L.addFirst(10);
L.first.next.next = L.first.next;
```

Access Control

```
public class SLList {
   public IntNode first;

   public SLList(int x) {
      first = new IntNode(x, null);
   }

   public void addFirst(int x) {
      first = new IntNode(x, first);
   }

   ...
}
```

We can prevent programmers from making such mistakes with the `private` keyword.

Improvement #3: Access Control

```java
public class SLList {
   private IntNode first;

   public SLList(int x) {
      first = new IntNode(x, null);
   }

   public void addFirst(int x) {
      first = new IntNode(x, first);
   }

   ...
}
```

Use the **private** keyword to prevent code in other classes from using members (or constructors) of a class.

```java
SLList L = new SLList(15);
L.addFirst(10);
L.first.next.next = L.first.next;
```

```
jug ~/Dropbox/61b/lec/lists2
$ javac SLListUser.java
SLListUser.java:8: error: first has private access in SLList
        L.first.next.next = L.first.next;
```

Why Restrict Access?

Hide implementation details from users of your class.

- Less for user of class to understand.
- Safe for you to change private methods (implementation).

Car analogy:

- **Public**: Pedals, Steering Wheel **Private**: Fuel line, Rotary valve

- Despite the term 'access control':
    - Nothing to do with protection against hackers, spies, and other evil entities.

Improvement #4: Nested Classes

Can combine two classes into one file pretty simply.

```
public class SLList {
   public class IntNode {
      public int item;
      public IntNode next;
      public IntNode(int i, IntNode n) {
         item = i;
         next = n;
      }
   }

   private IntNode first;
   public SLList(int x) {
      first = new IntNode(x, null);
   } ...
```

Nested class definition.

Could have made `IntNode` a private nested class if we wanted.

Instance variables, constructors, and methods of `SLList` typically go below nested class definition.

Why Nested Classes?

Nested Classes are useful when a class doesn't stand on its own and is obviously subordinate to another class.

- Make the nested class private if other classes should never use the nested class.

In my opinion, probably makes sense to make `IntNode` a nested private class.

- Hard to imagine other classes having a need to manipulate `IntNodes`.

Static Nested Classes

If the nested class never uses any instance variables or methods of the outer class, declare it static.

- Static classes cannot access outer class's instance variables or methods.
- Results in a minor savings of memory. See book for more details / exercise.

```
public class SLList {
    private static class IntNode {
        public int item;
        public IntNode next;
        public IntNode(int i, IntNode n) {
            item = i;
            next = n;
        }
    ...
```

We can declare IntNode static, since it never uses any of SLList's instance variables or methods.

Analogy: Static methods had no way to access "my" instance variables. Static classes cannot access "my" outer class's instance variables.

Unimportant note: For private nested classes, access modifiers are irrelevant.

addLast() and size()

Adding More SLList Functionality

To motivate our remaining improvements, and to give more functionality to our `SLList` class, let's add:

- `.addLast(int x)`
- `.size()`

See study guide for starter code!

Recommendation: Try writing them yourself before watching how I do it.

Methods	Non-Obvious Improvements	
`addFirst(int x)`	#1	Rebranding: `IntList` → `IntNode`
`getFirst`	#2	Bureaucracy: `SLList`
	#3	Access Control: `public` → `private`
	#4	Nested Class: Bringing `IntNode` into `SLList`

Answers not shown in slides. See sp19-lectureCode or video for answers.

Private Recursive Helper Methods

To implement a recursive method in a class that is not itself recursive (e.g. SLList):

- Create a private recursive helper method.
- Have the public method call the private recursive helper method.

```
public class SLList {
    private int size(IntNode p) {
        if (p.next == null) {
            return 1;
        }

        return 1 + size(p.next);
    }

    public int size() {
        return size(first);
    }
}
```

Efficiency of Size: http://yellkey.com/design

How efficient is size?

- Suppose size takes 2 seconds on a list of size 1,000.
- How long will it take on a list of size 1,000,000?

a. 0.002 seconds.
b. 2 seconds.
c. 2,000 seconds.
d. 2,000,000 seconds.

```java
public class SLList {
    private int size(IntNode p) {
        if (p.next == null) {
            return 1;
        }

        return 1 + size(p.next);
    }

    public int size() {
        return size(first);
    }
}
```

Improvement #5: Fast size()

Your goal:

- Modify SLList so that the execution time of size() is always fast (i.e. independent of the size of the list).

Also I have a gift giveaway: Accidentally some CDs from KALX 10 years ago. Come take one from the front if you want one.

```
private IntNode first;

public SLList(int x) {
  first = new IntNode(x, null);
}

public void addFirst(int x) {
  First = new IntNode(x, front);
}

private int size(IntNode p) {
  if (p.next == null)
    return 1;
  return 1 + size(p.next);
}

public int size() {
  return size(first);
}
```

Improvement #5: Fast size()

Solution: Maintain a special size variable that **caches** the size of the list.

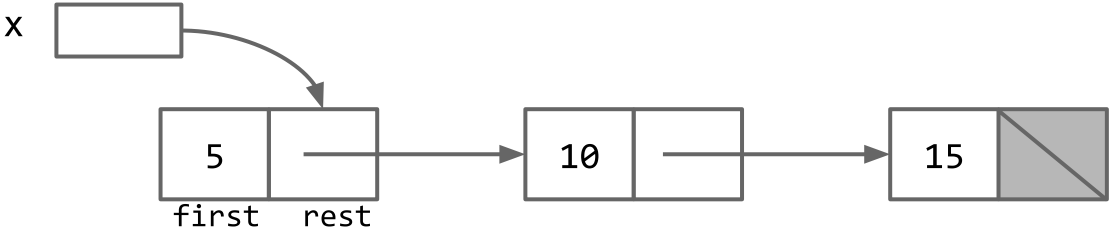
- Caching: putting aside data to speed up retrieval.

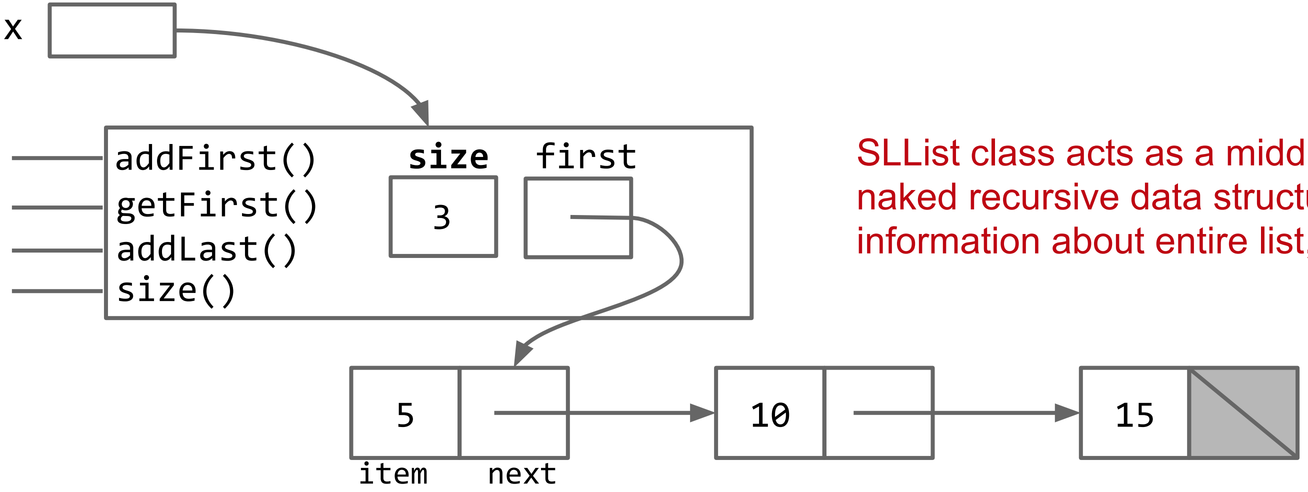
TANSTAAFL: There ain't no such thing as a free lunch.

- But spreading the work over each add call is a net win in almost any circumstance.

http://www.ensler.us/ensler.us/images/nolnchsmalla.jpg

Naked Linked Lists (IntList) vs. SLLists



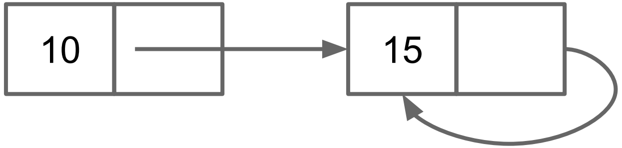


SLList class acts as a middle man between user and the naked recursive data structure. Allows us to store meta information about entire list, e.g. **`size`**.

Improvement #6a: Representing the Empty List

Benefits of `SLList` vs. `IntList` so far:

- Faster `size()` method than would have been convenient for `IntList`.
- User of an `SLList` never sees the `IntList` class.
  - Simpler to use.
  - More efficient `addFirst` method (see exercises).
  - Avoids errors (or malfeasance):



Another benefit we can gain:

- Easy to represent the empty list. Represent the empty list by setting `first` to null. Let's try!

How Would You Fix addLast?

Your goal:

- Fix addLast so that we do not get a null pointer exception when we try to add to the back of an empty SLList:

```
SLList s1 = new SLList();
s1.addLast(5);
```

See study guide for starter code if you want to try on a computer.

```
public class SLList {
   private IntNode first;
   private int size;

   public SLList() {
      first = null;
      size = 0;
   }

   public void addLast(int x) {
      size += 1;
      IntNode p = first;
      while (p.next != null) {
         p = p.next;
      }

      p.next = new IntNode(x, null);
   } ...
```

One Solution

One possible solution:

- Add a special case for the empty list.

But there are other ways...

```java
public void addLast(int x) {
  size += 1;

  if (first == null) {
    first = new IntNode(x, null);
    return;
  }

  IntNode p = first;
  while (p.next != null) {
    p = p.next;
  }

  p.next = new IntNode(x, null);
}
```

Sentinel Nodes

Tip For Being a Good Programmer: Keep Code Simple

As a human programmer, you only have so much working memory.

- You want to restrict the amount of complexity in your life!
- Simple code is (usually) good code.
  - Special cases are not ‘simple’.

```
public void addLast(int x) {
  size += 1;

  if (first == null) {
    first = new IntNode(x, null);
    return;
  }

  IntNode p = first;
  while (p.next != null) {
    p = p.next;
  }

  p.next = new IntNode(x, null);
}
```

addLast's Fundamental Problem

The fundamental problem:

- The empty list has a null **`first`**. Can't access `first.next`!

Our fix is a bit ugly:

- Requires a special case.
- More complex data structures will have many more special cases (gross!!)

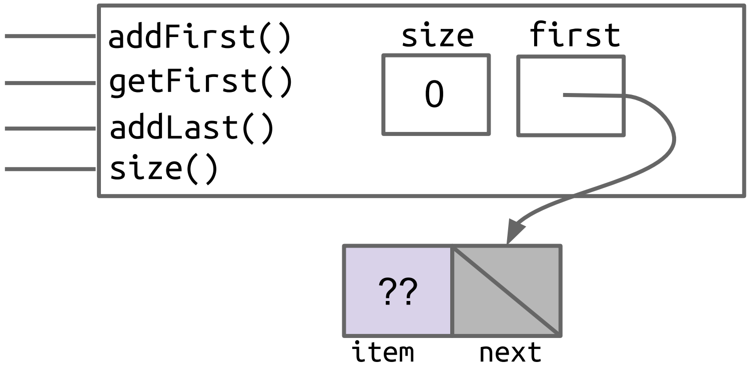
How can we avoid special cases?

- Make all `SLLists` (even empty) the "same".

```
public void addLast(int x) {
  size += 1;

  if (first == null) {
    first = new IntNode(x, null);
    return;
  }

  IntNode p = first;
  while (p.next != null) {
    p = p.next;
  }

  p.next = new IntNode(x, null);
}
```

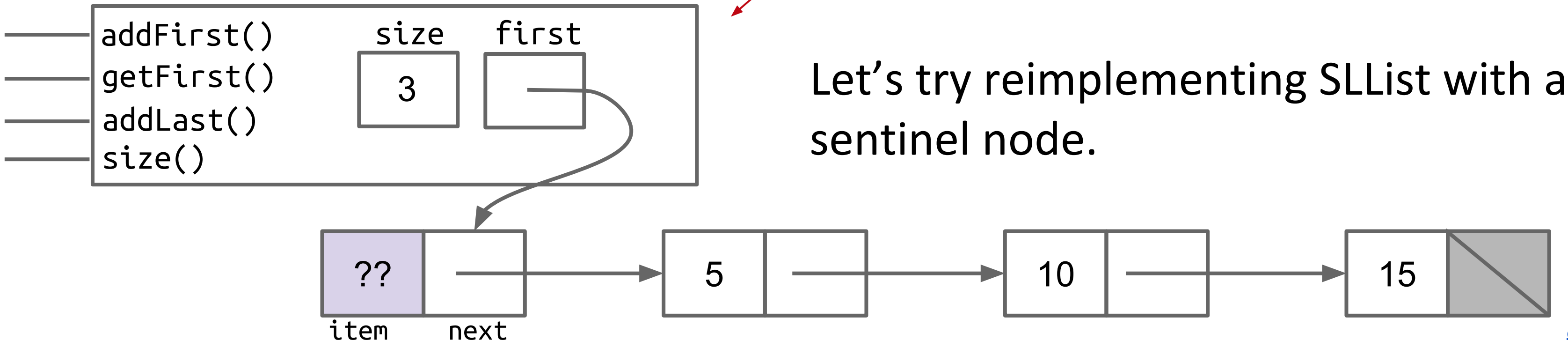
Improvement #6b: Representing the Empty List Using a Sentinel

Create a special node that is always there! Let's call it a "sentinel node".



The empty list is just the sentinel node.

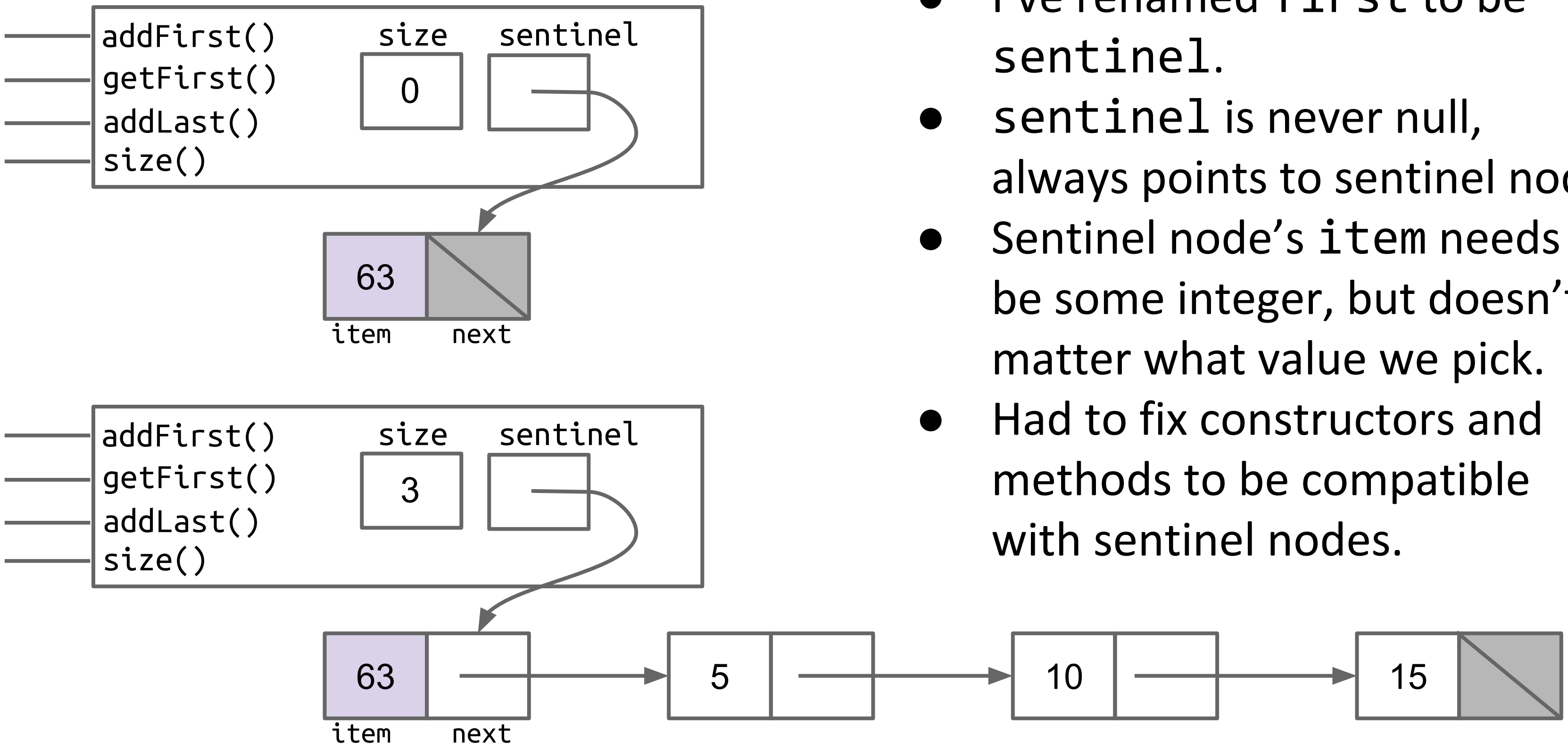
A list with 3 numbers has a sentinel node and 3 nodes that contain real data.



Let's try reimplementing SLList with a sentinel node.

Sentinel Node

The sentinel node is always there for you.



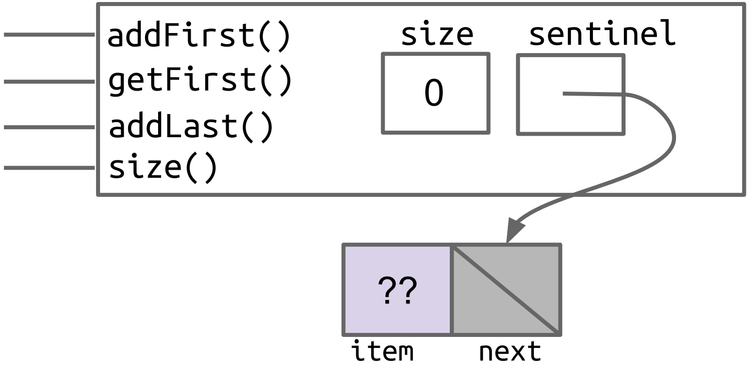
Notes:

- I've renamed `first` to be `sentinel`.
- `sentinel` is never null, always points to sentinel node.
- Sentinel node's `item` needs to be some integer, but doesn't matter what value we pick.
- Had to fix constructors and methods to be compatible with sentinel nodes.

addLast (with Sentinel Node)

Bottom line: Having a sentinel simplifies our `addLast` method.

- No need for a special case to check if `sentinel` is null (since it is never null).




```java
public void addLast(int x) {
  size += 1;

  if (sentinel == null) {
    sentinel = new IntNode(x, null);
    return;
  }

  IntNode p = sentinel;
  while (p.next != null) {
    p = p.next;
  }

  p.next = new IntNode(x, null);
}
```

Invariants

An invariant is a condition that is guaranteed to be true during code execution (assuming there are no bugs in your code).

An `SLList` with a sentinel node has at least the following invariants:

- The `sentinel` reference always points to a sentinel node.
- The first node (if it exists), is always at `sentinel.next`.
- The `size` variable is always the total number of items that have been added.

Invariants make it easier to reason about code:

- Can assume they are true to simplify code (e.g. `addLast` doesn't need to worry about nulls).
- Must ensure that methods preserve invariants.

Summary

Methods	Non-Obvious Improvements	
`addFirst(int x)`	#1	Rebranding: `IntList` → `IntNode`
`getFirst`	#2	Bureaucracy: `SLList`
`size`	#3	Access Control: `public` → `private`
`addLast(int x)`	#4	Nested Class: Bringing `IntNode` into `SLList`
	#5	Caching: Saving `size` as an `int`.
	#6	Generalizing: Adding a `sentinel` node to allow representation of the empty list.

For Those Who Were a Bit Bewildered!

Don’t panic if it felt fast!

The `LinkedListDeque` class that you’ll build in project 1 (to be officially released Friday) will give you practice so that you can deeply understand the ideas from today’s lecture.

Old Deprecated Slides

Improvement #7: Helper Methods

Suppose we wanted to write a `getBack()` method.

- Would be quite similar to `insertBack()`
- Make sense to create a `getBackNode()` method that can be used by both `getBack()` and `insertBack()`